Introduction to CUDA C

outline

- Introduction to CUDA C
- A First Program, Hello world
- A kernel call
- Passing parameters
- Querying devices
- Using device properties

A First Program

```
#include "../common/book.h"
int main( void )
{
    printf( "Hello, World!\n" );
      return 0;
}
```

-Example illustrates that there is no basic difference between Standard C and Cuda C

- This program runs entirely on Host

A First Program

- Host= CPU + System Memory
- Device= GPU+ its Memory
- A function that executes on the Device is Kernel

Contd..

```
#include<iostream.h>
_ _global_ _void kernel(void) {
}
Int main(void){
kernel<<<1,1>>>{};
printf(“Hello, World\n”);
return 0;
}
```

- The angle brackets denote arguments are passed to the runtime system.
- These are not arguments to the device code but are parameters that will influence how the runtime will launch our device code

Contd..

- CUDA C adds the __global__ qualifier to standard C.
- This mechanism alerts the compiler that a function should be compiled to run on a device instead of the host.
- Example: nvcc gives the function kernel() to the compiler that handles device code, and it feeds main() to the host compiler.
- Host code to one compiler and device code to another compiler

Passing Parameter

- #include "book.h“
- __global__ void add(int a, int b, int *c)
- { *c = a + b;
- }
- int main(void)
- {
- int c; int *dev_c;
- HANDLE_ERROR(cudaMalloc((void**)&dev_c, sizeof(int)));
- add<<1,1>>(2, 7, dev_c);
- HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost));
- printf("2 + 7 = %d\n", c);
- cudaFree(dev_c);
- return 0;
- }

Contd.

- Pass parameters to a kernel as done with any C function.
- Allocate memory to do anything useful on a device, such as return values to the host.
- Angle-bracket syntax notwithstanding, a kernel call looks and acts exactly like any function call in standard C.
- The runtime system takes care of any complexity introduced by the fact that these parameters need to get from the host to the device

Contd.

- HANDLE_ERROR(): calls is a utility macro, It simply detects that the call has returned an error, prints the associated error message, and exits the application with an EXIT_FAILURE code.

- cudaMalloc():allocation of memory, but it tells the CUDA runtime to allocate the memory on the device.
- cudaMalloc((void**)&dev_c, sizeof(int))
- The first argument is a pointer to the pointer you want to hold the address of the newly allocated memory, and the second parameter is the size of the allocation you want to make.

Contd..

- The compiler cannot protect you from this mistake, either. It will be perfectly happy to allow dereferences of device pointers in your host code because it looks like any other pointer in the application. We can summarize the restrictions on the usage of device pointer as follows:
- Pass pointers allocated with cudaMalloc() to functions that execute on the device.
- Use pointers allocated with cudaMalloc()to read or write memory from code that executes on the device.
- Pass pointers allocated with cudaMalloc()to functions that execute on the host.
- Cannot use pointers allocated with cudaMalloc()to read or write memory from code that executes on the host

Contd..

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- cudaMemcpy() :memory on a device through calls to from host code.
- These calls behave exactly like standard C memcpy() with an additional parameter to specify which of the source and destination pointers point to device memory.
- In the example, notice that the last parameter to cudaMemcpy() is cudaMemcpyDeviceToHost, instructing the runtime that the source pointer is a device pointer and the destination pointer is a host pointer

Contd..

- cudaMemcpyHostToDevice would indicate the opposite situation, where the source data is on the host and the destination is an address on the device.
- Finally, we can even specify that both pointers are on the device by passing cudaMemcpyDeviceToDevice.
- If the source and destination pointers are both on the host, we would simply use standard C’s memcpy() routine to copy between them

Querying devices

- cudaGetDeviceCount(): to know how many devices in the system were built on the CUDA architecture. These devices will be capable of executing kernels written in CUDA C.
- int count;
  HANDLE_ERROR(cudaGetDeviceCount(&count));
- cudaDeviceProp structure returns(see from the textbook)

Device Properties

- cudaDeviceProp prop; memset(&prop, 0, sizeof(cudaDeviceProp)); prop.major = 1; prop.minor = 3
- cudaGetDeviceCount() and cudaGetDeviceProperties(), we could iterate through each device and look for one that either has a major version greater than 1 or has a major version of 1 and minor version greater than or equal to 3

Contd..

- cudaChooseDevice() to have the CUDA runtime find a device that satisfies this constraint. The call to cudaChooseDevice() returns a device ID that we can then pass to cudaSetDevice().

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp  prop;
    int dev;

    HANDLE_ERROR( cudaGetDevice( &dev ) );
    printf( "ID of current CUDA device:  %d\n", dev );

    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
    printf( "ID of CUDA device closest to revision 1.3:  %d\n", dev );
    HANDLE_ERROR( cudaSetDevice( dev ) );
}
```

```
#include <stdio.h>

// Print device properties
void printDevProp(cudaDeviceProp devProp)
{
    printf("Major revision number:         %d\n",  devProp.major);
    printf("Minor revision number:         %d\n",  devProp.minor);
    printf("Name:                          %s\n",  devProp.name);
    printf("Total global memory:           %lu\n",  devProp.totalGlobalMem);
    printf("Total shared memory per block: %lu\n",  devProp.sharedMemPerBlock);
    printf("Total registers per block:     %d\n",  devProp.regsPerBlock);
    printf("Warp size:                     %d\n",  devProp.warpSize);
    printf("Maximum memory pitch:          %lu\n",  devProp.memPitch);
    printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of block:  %d\n", i, devProp.maxThreadsDim[i]);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
    printf("Clock rate:                    %d\n",  devProp.clockRate);
    printf("Total constant memory:         %lu\n",  devProp.totalConstMem);
    printf("Texture alignment:             %lu\n",  devProp.textureAlignment);
    printf("Concurrent copy and execution: %s\n",  (devProp.deviceOverlap ? "Yes" :
"No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout:      %s\n",  (devProp.kernelExecTimeoutEnabled
?"Yes" : "No"));
    return;
}

int main()
```

```
    printf("Total constant memory:          %lu\n",  devProp.totalConstMem);
    printf("Texture alignment:              %lu\n",  devProp.textureAlignment);
    printf("Concurrent copy and execution: %s\n",  (devProp.deviceOverlap ? "Yes" :
"No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout:      %s\n",  (devProp.kernelExecTimeoutEnabled
?"Yes" : "No"));
    return;
}

int main()
{
    int devCount;
    cudaGetDeviceCount(&devCount);
    printf("CUDA Device Query...\n");
    printf("There are %d CUDA devices.\n", devCount);

    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printDevProp(devProp);
    }
    return 0;
}
```

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         3
Minor revision number:         5
Name:                          Tesla K20m
Total global memory:           5032706048
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    705500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
```